Dalhousie University
Department of Electrical and Computer Engineering

ECED 3403 – Computer Architecture

Testing requirements with examples

Justin Lynch and Larry Hughes

1 June 2018

1 Introduction

Testing shows the presence, not the absence, of bugs[1]

The computer engineer must thoroughly test, even with unlikely parameters, the hardware and software, and ultimately the system itself, to ensure that the system operates properly and reliably.[2]

Software testing is critical to the functioning of any system. It is the verification of the design. Handling common, and predictable cases is just the start of well-planned software. The solution must handle a reasonable range of inputs that may have unexpected behaviours.

A test may explore an *edge case*. An edge case represents a scenario caused within the code that may stretch a data type beyond its capacity such as an input string or location counter. How large can the location counter be? What happens when a counter is incremented past the largest possible value? Is this acceptable or predictable?

Certain *special cases* may produce unintended input; for example, escape characters such as '\n' or '\t' have different meanings in strings.

Testing should consider a reasonable subset of inputs, outputs, and ranges that would push the code to its limits. The tests must stretch the code and ensure desired functionality.

Design the tests carefully and decide what is to be tested. Broadly speaking, there are four levels of testing:[3]

Unit Testing: A level of the software testing process where individual units of a software are tested. The purpose is to validate that each unit of the software performs as designed.

Integration Testing: A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

System Testing: A level of the software testing process where a complete, integrated system is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.

[1] Dijkstra (1969) J.N. Buxton and B. Randell, eds, *Software Engineering Techniques*, April 1970, p. 16. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969.

[2] *Computer Engineering Curricula 2016*. Retrieved June 22, 2017, from Association for Computer Machinery - Curricula Recommendations: http://www.acm.org/binaries/content/assets/education/ce2016-final-report.pdf

[3] *Software Testing Levels*, Software Testing Fundamentals, http://softwaretestingfundamentals.com/software-testing-levels/. Accessed 1 June 2018

Acceptance Testing: A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Regardless of the level, if a test fails, it is necessary to identify *why* it failed. Keep in mind that there are essentially three possible reasons for a test failing:

1. The design was incorrect (i.e., the designer didn't understand the problem)
2. The implementation was incorrect (i.e., the coder didn't understand the design)
3. The test was incorrect (the tester didn't understand what was to be tested)

If a test fails, examine the software to determine why it failed and what corrections are necessary to have it to pass the tests. It is necessary to properly identify what is being tested, and ensure the test actually verifies it.

Does the test do what is expected of it? Maybe the input file has hidden characters that aren't being accounting for? Or maybe the program was initialized in a state that can't transition to the state to be tested?

Sometimes the test is well thought out but there were problems with implementation. Perhaps the value of a pointer is printed which eventually gets set to a NULL value. On the last iteration, the code may throw a segmentation fault. In this case, the printing of the pointer within the test caused the error, not the code itself.

If a test is to be changed, be sure to justify and understand what was wrong with it. Do not change tests just to make the code pass.

It is important to remember that when correcting a fault in the software, the overall structure of the software must be taken into account. A correction for one failed test may break another. Be sure to re-run existing tests, and ensure the correction didn't cause a fault elsewhere.

2 Requirements

In ECED 3403, you are required to perform a number of system tests to demonstrate that your software meets these test requirements. Each test is to have the following structure:

Name: Tests need an identifier. The name should give an indication as to what was tested. Reading the names of your test should give the reader a good idea of the capability of your software

Purpose/Objective: Be detailed. Identify the potential problem, the defined behavior when encountering this problem, the inputs to test the case, and the expected outputs. The exact scope of the test and what it's doing should be clear.

Test Configuration: Not all tests start from scratch. Sometimes, it is reasonable to assume a certain state or limited type of input to narrow the test focus. This section should include all the relevant inputs, settings, configurations and assumptions made during the test.

Expected Results: The expected results of the test.

Actual Results: A capture of the output (i.e., actual results) of the program.

By rights, the test requirements should be written independently of the software, often at the design stage. In other words, the programmer shouldn't run the program and use the results as the actual results.

3 Examples

Two examples of software testing are given in this section.

3.1 Example 1

Software to handle pedestrian and vehicular traffic at an intersection has been designed. A series of tests have been designed; this one deals with the northbound and southbound light activation.

Assume that nLane and sLane are defined in the data dictionary as follows (note, more detailed descriptions would be required):

nLane = red + yellow + green + turn + walk
sLane = red + yellow + green + turn + walk

TEST 1: Walk Light On Advance Green

Purpose: This test is to ensure that when transitioning from north and south lanes stopped, to a north lane advance green, that only the north traffic and walk lights change, while the south stays red.

Test Configuration: The initial state of the traffic light system will be no walk light, and a red stop light for both the northbound and southbound lanes. Time proceeds in increments of 5. At time = 10, the northbound lights will change.

Initial state of lights:

```
nLane.red = true;
sLane.red= true;
sLane.yellow = false;
nLane.yellow = false;
nLane.walk = false;
sLane.walk = false;
nLane.turn = false;
sLane.turn = false;
nLane.green = false;
sLane.green = false;
```

Expected results: When the program is run, the expected results are as follows:

Timer	**Northbound**					**Southbound**				
	Red	Yellow	Green	Turn	Walk	Red	Yellow	Green	Turn	Walk
0	1	0	0	0	0	1	0	0	0	0
5	1	0	0	0	0	1	0	0	0	0
10	0	0	1	1	1	1	0	0	0	0
15	0	0	1	1	1	1	0	0	0	0

Actual results: The lights start off with red illuminated. At 10 seconds, as predicted, the northbound lane turns green, with a green advance arrow, and pedestrian traffic allowed. The southbound lane remains at red. The actual results are as follows:

```
Timer = 0
Northbound Red, yellow, green, turn, walk: 1 ,0 ,0 ,0 ,0
Southbound Red, yellow, green, turn, walk: 1 ,0 ,0 ,0 ,0
Timer = 5
Northbound Red, yellow, green, turn, walk: 1 ,0 ,0 ,0 ,0
Southbound Red, yellow, green, turn, walk: 1 ,0 ,0 ,0 ,0
Timer = 10
Northbound Red, yellow, green, turn, walk: 0 ,0 ,1 ,1 ,1
Southbound Red, yellow, green, turn, walk: 1 ,0 ,0 ,0 ,0
Timer = 15
Northbound Red, yellow, green, turn, walk: 0 ,0 ,1 ,1 ,1
Southbound Red, yellow, green, turn, walk: 1 ,0 ,0 ,0 ,0
```

The above shows, all lights are red. At timer = 10, the northbound light turns red off, green on, turn on and walk on. The southbound light stays red, as expected. The test passes.

3.2 Example 2

A program has been written to add two numbers and print the result. The program accepts pairs of numbers and display the result of the addition; this is repeated five times.

TEST 1: Positive number tests

Purpose: This test supplies the program with a range of positive numbers.

Configuration: The following pairs of numbers are to be supplied after the prompt "Enter two numbers to add":

```
1 2
99 1
5 10
100 100
100 50
```

Expected results: The expected results are:

```
Result of 1 + 2 is: 3
Result of 99 + 1 is: 100
Result of 5 + 10 is: 15
Result of 100 + 100 is: 200
Result of 100 + 50 is: 150
```

Actual results: The results from the program were:

```
Result of 1 + 2 is: 3
Result of 99 + 1 is: 100
Result of 5 + 10 is: 15
Result of 100 + 100 is: -56
Result of 100 + 50 is: -106
```

The first three tests are correct, while the final two are incorrect. In addition, when the program terminates, there is a diagnostic: "*Run-Time Check Failure #2 – Stack around the variable 'v1' was corrupted.*"

The test fails. The results are incorrect.

<u>TEST 2: Negative number tests</u>

Purpose: This test supplies the program with a range of negative numbers.

Configuration: The following pairs of numbers are to be supplied after the prompt "Enter two numbers to add":

```
-1 -1
-100 -50
-128 -1
-127 -1
-256 0
```

Expected results:

```
Result of -1 + -1 is: -2
Result of -100 + -50 is: -150
Result of -128 + -1 is: -129
Result of -127 + -1 is: -128
Result of -256 + 0 is: -256
```

Actual results:

```
Result of -1 + -1 is: -2
Result of -100 + -50 is: 106
Result of -128 + -1 is: 127
Result of -127 + -1 is: -128
Result of 0 + 0 is: 0
```

The first test is correct, the remaining four tests are incorrect. In addition, when the program terminates, there is a diagnostic: "*Run-Time Check Failure #2 – Stack around the variable 'v1' was corrupted.*"

The test fails. The results are incorrect.

From these two tests and the associated results, it is clear that the program does not work; *but the tests worked – they showed that something is wrong with the software*.

Here is the code. Based on the difference between the actual and expected results, can you correct the fault? Once corrected, how will you know that the fault has been corrected?[4]

```
/*
 Program to prompt a user for pairs of numbers, add them, and display the results
 A. Nonymous
 1 June 2018
*/
```

[4] The correct answer is, run the tests again. In other words, don't assume just because a correction has been made that the software works. Each time a correction is made, the tests must be run again – both to show that the correction worked *and* to show that other faults weren't added. There is no need to run the program with the debugger, you should be able to solve the problem from the information given.

```
#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>

void main()
{
char v1, v2, res;    /* Input values and result */
int i;               /* Loop counter*/

for (i=0; i<5; i++)
{
       printf("Enter two numbers to add:\n");
       scanf_s("%d %d", &v1, &v2);
       res = v1 + v2;
       printf("Result of %d + %d is: %d\n", v1, v2, res);
       getchar(); /* Remove CR */
}
getchar(); /* Give the user time to examine the output */
}
```